Assignment 1 – Introduction to Socket Programming

CS456/656 Computer Networks

Spring 2020

1) Objective

The goal of this assignment is to gain experience with socket programming and to implement a simple message board with a client and a server. The message server will store a list of messages and, upon request send those messages to a client. First, the server and client will negotiate, using TCP, a port over which all other communications will take place. Then, the server will send the list of stored messages, over UDP, to the client who will display them. Next, the client will send a message to the server, which will be added to the list of stored messages at the server.

2) Background

You will find detailed description for basic socket programming in python using both TCP and UDP in Chapter 2, Sections 2.7 Socket Programming: Creating Network Applications, 2.7.1 Socket Programming with UDP and 2.7.2 Socket Programming with TCP.

3) Specifications

This assignment uses a two-stage communication process. The first stage is negotiation followed by a transaction stage. In the negotiation stage, the client will connect to the server using the server address and predefined port number, to negotiate a port number for future communications in the transactions stage. Once the port number for transactions stage has been confirmed, the client will communicate with server to retrieve and display the list of messages and post messages to the server.

a) Negotiation Phase communication over TCP
   i) Client creates a TCP connection with the server using `<server_address>` and `<n_port>`, the server address and negotiation port number of the server, respectively.
   ii) The client sends a request to get a random port number from the server. To initiate this negotiation, the client sends a request code `<req_code>`, an integer.
   iii) The server receives the request code from the client and verifies it
      - If `<req_code>` is invalid, the server will reply with a random port number `<r_port>` of `0` and terminate the TCP connection. If a client receives `<r_port>` value of `0`, it should terminate with an error "`Invalid req_code.`"
      - If `<req_code>` is valid, the server will reply with a random port number `<r_port>`, for transaction stage. After receiving a valid `<r_port>` the client closes the TCP connection with the server.
   iv) The server continues listening on its `<n_port>` for subsequent client requests

b) Transaction Phase communication over UDP
   i) The client sends a message "`GET`," to the server.

ii) The server should then send the list of stored messages back to the client
   - When there are no more messages, the server sends "`NO MSG`" to the client

iii) The client will display each message on its own line
iv) The client sends its text message `<msg>` to the server.
v) The server stores the message as shown below

```
[r_port]: <msg>
```

vi) If the client sends the message "TERMINATE," the server shuts down.
vii) The client waits for keyboard input before exiting.

4) Deliverables

a) **A client program (client)**

Implement a program, client, which will send and receive messages from the message server. The client program takes four parameters, in order, `<server_address>`, `<n_port>`, `<req_code>`, and `<msg>`. Where `<server_address>` is address of server, `<n_port>` is the port used in the negotiation stage, `<req_code>` is the validation code for receiving a valid port number for the transaction stage and `<msg>` is the text message sent to the server to add to the list of messages or terminate.

The client should send the "`GET`" command to the server to retrieve list of stored messages from the server and print the list of messages, one per line. Note, there will only be multiple messages if multiple clients have been run prior to this execution. Afterward, the client should send `<msg>` to the server. If `<msg> == TERMINATE`, then the server should shut down. The client waits for keyboard input before exiting.

b) **A client log file**

For grading purposes, you should maintain a file named "`client.txt`" and your client program should write the line `<r_port>` to that file with the messages in order.

Please note that the client log file must not be overwritten if it already exists. It **must only be appended to** if it exists or created if it does not exist.

c) **A server program (server)**

Implement a program, named server, which will be the message server. The server requires a single parameter, `<req_code>`, which is used to validate clients attempting to connect. When the server starts, it must print out the negotiation port number `<n_port>` in the following format, as the first line.

```
[SERVER_PORT]: <n_port>
```

For example, if the negotiation port of the server is 52500, then the server should print: SERVER_PORT: 52500.

The server must accept multiple, concurrent connections. It must store all of the messages from every client. Each client and its message should be stored in the following format:

```
[CLIENT_PORT]: <msg>
```

The `[CLIENT_PORT]` should be the `<r_port>` used by the that client to communicate with the server during the transaction stage of that client-server communication. The server should send the list of all messages to a client if the client sends a `<msg> == GET`. If there are no stored messages, or after all the messages have been sent the server sends a "`NO MSG`" message to the client. Note, the server should not remove any messages that it stores.

d) **A server log file**
For grading purposes, in addition to printing `[SERVER_PORT]: <n_port>` to the screen, your server program should also write the line `[SERVER_PORT]: <n_port>` to a file named "`server.txt.`"

5) Example Execution

Two shell scripts named `server.sh` and `client.sh` are provided. Modify them according to your choice of programming language. You should execute these shell scripts which will then call your client and server programs.

@server s

```
./server.sh <req_code>
```

@client

```
./client.sh <server_address> <n_port> <req_code> 'first socket
program'
```

6) Sample Scenario

The scenario outlined below has been designed with a single client and a server program in mind. Consider a scenario where the client wants to send 2 messages to a server one after the other followed by a "TERMINATE" message. The following steps are part of such a scenario with the output from the client side and any log files detailed in each step.

I. Server Execution:

a.
```
./server.sh 1221
```

b. The server is launched here with a **valid request code** of ***1221***. Any other request code used by the client should be treated as invalid.

c. The server should obtain a free port from the OS for use as it's <n_port>. It should write this info to a file named server.txt as mentioned above.
    i. For the case of this scenario, let us assume that this **<n_port>** has a value of ***5432***.
    ii. A sample "server.txt" file has this output in the format shown below

```
SERVER_PORT: 5432
```

d. Let us also assume that the **server** is running at ***192.168.1.45*** for this scenario

II. Client Execution – First Iteration

a. The client needs to be launched with a valid <server_address>, <n_port> and <req_code> as shown below

```
./client.sh 192.168.1.45 5432 1221 'Message 1'
```

b. The client first negotiates a <r_port> with the server. Let us assume that this **<r_port>** had a value of ***1111*** here.
c. The server has no prior messages stored and will return "No MSG" to the client. Thus, the client output is as shown below:

```
No MSG
```

d. The client needs to write this along with its <r_port> to a file named "client.txt" as mentioned above. This file is newly created since it does not already exist. After this, "client.txt" will be as shown below

```
r_port: 1111
No MSG
```

e. The client sends its message ("Message 1") to the server and then exits.

III. Client Execution – Second Iteration

a. The client needs to be launched with a valid <server_address>, <n_port> and <req_code> as shown below

```
./client.sh 192.168.1.45 5432 1221 'Message 2'
```

b. The client again negotiates a new <r_port> with the server. Let us assume it is ***2222*** in this iteration.
c. The server returns a list of the messages it has (with the correct <r_ports>) ending with the "No MSG" message. The client's output is shown below:

```
1111: Message 1
No MSG
```

d. The client **appends** this output to client.txt since the file already exists. The file will look as shown below after this iteration.

```
r_port: 1111
No MSG

r_port: 2222
1111: Message 1
No MSG
```

e. The client sends its message ("Message 2") to the server and exits.

IV. Client Execution – Final Iteration

a. The client needs to be launched with a valid <server_address>, <n_port> and <req_code> as shown below

```
./client.sh 192.168.1.45 5432 1221 'TERMINATE'
```

b. The client again negotiates a new <r_port> with the server. Let us assume it is ***3333*** in this iteration.
c. The server returns a list of the messages it has (with the correct <r_ports>) ending with the "No MSG" message. The client's output is shown below:

```
1111: Message 1
2222: Message 2
No MSG
```

d. The client **appends** this output to client.txt since the file already exists. The file will look as shown below after this iteration.

```
r_port: 1111
No MSG

r_port: 2222
1111: Message 1
No MSG

r_port: 3333
1111: Message 1
2222: Message 2
No MSG
```

e. The client sends the string "TERMINATE" to the server and exits. This will also cause server termination.

7) Hints

- You can use the python sample codes in Section 2.7 from the textbook.
- Use ports greater than 1024, since ports 0-1023 are reserved for other well-known applications.
- If there are problems establishing connections, check whether any of the computers running the server and the client is behind a firewall, the ports may be blocked.
- Make sure that the server is running before you run the client.
- Also remember to print the <n_port> where the server will be listening and make sure that the client is trying to connect to that same port for negotiation.
- If both the server and the client are running on the same system, 127.0.0.1 or localhost can be used as the destination host address.
- You have to ensure that both `<n_port>` and `<r_port>` are available. Just selecting a random port does not ensure that the port is not being used by another program.
- You could implement the server either to return a list of messages to the client at once or one message at a time.
    - Think about which would be better for server performance

8) Instructions for submission

Submit all your files in a single compressed file (.zip, .tar etc.) using Assignment 1 drop box in LEARN. You may use guides, books, etc., as long as you cite your sources appropriately. But remember, you cannot share your program or work with any other student or in groups. This programming assignment is to be completely individually. You can use any programming language to design and implement the and server programs. You are expected to have a reasonable amount of code documentation, to help the graders read your code. You **will** lose marks if your code is not readable, or documented.

You must hand in the following files / documents:

a) all source code files
b) Makefile, your code **must** compile and link by typing "`make`" or "`gmake`".

c) README file: this file **must** contain instructions on how to run your program, which undergrad machines your program was built and tested on, and what version of *make* and *compilers* you are using.
d) Make sure that no additional (manual) input is required to run the server or client programs.
e) Modified server.sh and client.sh scripts
f) All codes must be tested in the `linux.student.cs` environment prior to submission
   - Run client and server in two different `student.cs` machines
   - Run both client and server in a single `student.cs` machine

Your implementation will be tested on the machines available in the **undergrad environment** https://cs.uwaterloo.ca/cscf/internal/infrastructure/inventory/CS-TEACHING/hosts.
Please compile and test your code on these machines prior to submission.

The late policy is a 10% penalty for each day (24hrs) late up to a maximum of 3 days. Submissions will not be accepted beyond 3 late days.

9) Grading Rubric

No	Scenario Description	Expected Response	Failure Penalty
1	**Client and Server Code Compilation**	Code compiles correctly	Depends on severity
2	**Single Client and Server on the same machine** • Client: Sends incorrect server code	Client terminates by printing "Invalid `req_code`"	5%
3	**Single Client and Server on the same machine** • Client Message: "TERMINATE"	**Client:** Client prints "`NO MSG`" and terminates. **Server:** Server terminates.	10%
4	**Single Client and Server on the same machine** • Client Message: "Message 1" • Client Message: "TERMINATE"	**Client:** • Print "`NO MSG`" on the first iteration • Print "Message 1" with <r_port> followed by "`NO MSG`" in the second iteration (when sending "TERMINATE"). **Server:** Terminates at the end.	15%
5	**Single Client and Server on the same machine** • Client Message: "`Message 1`" • Client Message: "`Message 2`" • Client Message: "`Message 3`" • Client Message: "`TERMINATE`"	**Client:** Each client iteration should print all the previous messages sent and the final iteration (when sending "TERMINATE") should print "`Message 1`", "`Message 2`",	15%

No	Scenario Description	Expected Response	Failure Penalty
		"`Message 3`" (along with the correct <r_ports>) and "`NO MSG`" in separate lines **Server:** Terminates at the end.	
6	**2 Clients and a Server, all on the same machine** • Note: Client 2 must be started after Client 1 exits • Client 1 Message: "Message from C1" • Client 2 Message: "Message from C2" • Client 2 Message: "TERMINATE"	**Client 1:** Client must print "NO MSG" and exit. **Client 2:** • Client must print only "Message from C1" (with <r_port>) and "NO MSG" in the first iteration. • Client must print "Message from C1", "Message from C2" (with the correct <r_ports>) and "NO MSG" on separate lines in the second iteration (when sending "TERMINATE") **Server:** Terminates at the end.	10%
7	**Single Client and Server on different machines** • Client Message: "Message 1" • Client Message: "TERMINATE"	**Client:** • Print "NO MSG" on the first iteration • Print "Message 1" (with <r_port>) followed by "NO MSG" in the second iteration (when sending "TERMINATE"). **Server:** Terminates at the end.	15%
8	**2 Clients and a Server, all on different machines** • Note: Client 2 must be started after Client 1 exits • Client 1 Message: "Message from C1" • Client 2 Message: "Message from C2" • Client 2 Message: "TERMINATE"	**Client 1:** Client must print "NO MSG" and exit. **Client 2:** • Client must print only "Message from C1" (with <r_port>) and "NO MSG" in the first iteration. • Client must print "Message from C1", "Message from C2" (both with the correct <r_ports>) and "NO MSG" on separate lines in the second iteration (when sending "TERMINATE.	10%

No	Scenario Description	Expected Response	Failure Penalty
		Server: Terminates at the end.	
9	**2 Clients and a Server, all on different machines** • Note: Each client instance must be started after the previous instance exits • Client 1 Message: "Message from C1" • Client 2 Message: "Message from C2" • Client 2 Message: "TERMINATE" • Client 1 Message: "The geese master-minded COVID-19"	**Client 1:** • Client must print "NO MSG" and exit when sending "Message from C1". • Client must timeout or exit because the server is down. **Client 2:** • Client must print only "Message from C1" (with the correct <r_port>) followed by "NO MSG" on the first iteration. • Client must print "Message from C1" and "Message from C2" (with the correct <r_ports>) followed by "No MSG" on separate lines on the second iteration (when sending "TERMINATE"). **Server:** Terminates after Client 2's instruction to do so.	10%
10	**Coding Style and Documentation**		5%

10) Additional Notes:

- **Compilation Failures**
  - If code compilation fails and can be fixed with minor changes, penalty is 5% on top of the scenarios above
  - If code compilation fails and you need to spend more than 10 minutes figuring out the issue / multiple issues, deduct 100%
- **Server Termination Failures**
  - If server termination does not occur, it is likely that both Scenarios 3 and 9 will fail and the termination clause for the other scenarios will as well.
  - In this case, deduct marks only for Scenarios 3 and 9 for a total of 20% and ignore the server termination failures in other scenarios.